An OS And Other Tools
For The SBC-85 Project

When microcomputers first appeared there was, of course, precious little software available for them. If you wanted to use a microcomputer for some particular purpose you were probably going to have to write the software yourself. That was the position I was in more than forty years ago with my first computer, an Explorer 85 from Netronics Research.

Explorer 85

My long-term goal had been to expand the system enough to eventually use CP/M, which was becoming the most popular OS for 8 bit computers, and for which there was a rapidly expanding wealth of software. During that time I was involved in deploying systems for the company I worked for that included magnetic bubble memory. When I learned that Intel was selling bubble memory evaluation kits I decided to use one of those for non-volatile storage instead of a more expensive 8 inch floppy drive. In preparation for building a floppy drive interface card, I had already purchased a copy of the book "K2FDOS, A Floppy Disk Operating System For the 8080" by Kenneth B. Welles (CP/M would have been a later upgrade). It took a lot of hours, but I eventually had K2FDOS working on my system, along with an editor/assembler from another book: "TEA, An 8080/8085 Co-Resident Editor/Assembler" by Christopher A. Titus. I also had another assembler that I had written from scratch that could assemble source code split into multiple files (RAM was expensive). I used that system for a long time, eventually replacing it with something newer, but I never actually got rid of it.

Last summer I discovered the SBC-85 project here, and decided to build one, thinking that perhaps I could get the software I had been using forty years ago to run on that hardware. The first thing to do was to build the SBC-85 system, including the CPU board, Memory Expansion board, Backplane, and the Bubble Memory board. I also built the Bus Monitor for good measure. It's a very nice system, but not all that useful without software, although it does have a very capable monitor program.

SBC-85 With Bubble Memory
And Bus Monitor Supreme

The next step was to try to recover as much software as possible from the old Explorer. In the end I was reasonably successful with both the bubble memory and cassette tapes. Now began the final stage of this project: getting that software to work on the SBC-85. K2FDOS is actually five separate programs: Boot, Sysgen, Format, PIP, and FDOS. The boot program reads an executable copy of FDOS into RAM. The sysgen program basically does the opposite: it saves the copy of FDOS that's in RAM. The format program prepares the storage media, whatever it is, for use by FDOS. PIP is an auxiliary program for FDOS that provides a user interface to perform common tasks like renaming, deleting, listing files, etc. And FDOS itself provides all the basic file and I/O functions that external programs might need.

Although I could find the source for most of the software I was interested in, the only source files I could find for FDOS itself were very early copies that didn't include any of the changes necessary to get it to work with bubble memory. However, I did have the executable and a printout that was nearly the same as the executable, so I used DASMx to disassemble the executable, resulting in a source file that assembled to a binary that matched the executable from the Explorer. For a cross assembler I use AS (or ASL on FreeBSD). It still took some effort to get that version of FDOS working on the SBC-85. For one thing, I had been using a parallel port on the Explorer as console in, whereas the SBC-85 uses the processor's SID pin. There were also a number of places in the source that didn't look quite right. Mostly cases where there were pointers referencing locations that didn't seem to make any sense. I had to go through all of those and try to figure out what the original intent was. Eventually I got everything fixed up.

The Intel bubble memory modules organize data into 2048 pages of 68 bytes each (64 bytes when using error correction), and the support chips provide a fairly easy to use interface. Original FDOS organizes 8 inch floppies into blocks of two sectors each, 16 blocks per track, and 77

tracks, and it included code for even the lowest level operations, including sector headers and checksums. For the bubble memory, I used blocks of 2 pages, 16 blocks per track, and 64 tracks, and I could skip all the really low-level functions. I reserved the first two tracks for an executable copy of FDOS, and the next two tracks for the bitmap and directory. The bitmap is used to keep track of which blocks were used and which were available. Each block included 21 bytes of overhead, which resulted in 115 bytes of usable data, compared to 256 bytes for original FDOS and 8 inch floppies. The bitmap thus needed to keep track of 60 tracks times 16 blocks per track, or 960 blocks. That meant I needed 120 bytes to store the bitmap (960/8 bits). But there were only 115 bytes available in one block, and I really didn't want to have to deal with the complexity of using two blocks for the bitmap. So I cheated, and "stole" 5 extra bytes. Those bytes would normally have been used by FDOS, so I inserted checks to avoid overwriting bitmap data. That turned out to be surprisingly easy.

The directory also needed a little attention. Each entry consists of 9 bytes for the filename (6.3 format), 2 bytes for initial track & sector, 2 bytes for the date, 1 byte for the size in blocks, and 2 bytes unused. With 16 bytes per entry, that results in 7 entries per block, and 112 per track. I had two tracks minus one block available, for a total of 217 entries. I could probably have easily gotten by with one track for the bitmap and directory, but I decided to leave things as they were. Besides, increasing the number of data blocks would have also increased the size of the bitmap, which would have been difficult. Although there's apparently just one byte for file size, there are actually two additional bits available in the date, so the maximum file size is 1024 blocks, or 117760 bytes. There were also some corrections that were needed for the routines that translated between bubble page numbers and track/sector, but those weren't very complicated. Very little needed to be done for the other four programs, other than what was necessary to make the source compatible with AS syntax.

Having an OS for the hardware still isn't enough though, you need some additional tools, like a text editor and an assembler. This is where the bulk of the effort went. I did have source files for the previously mentioned TEA editor and assembler, but it didn't look like I had the most recent version. Or perhaps I never did get a complete implementation finished. My job at the time took me out of town every week, so I only had weekends to tinker.

TEA was written at a time when Teletypes with paper tape readers and punches were used fairly often for terminals, and incorporated six I/O functions: console status, console in, console in with echo, console out, reader in, and punch out. The TEA book had clear explanations of exactly how all the I/O functions needed to work, so adapting the console functions to FDOS was fairly easy. You would think that it would also be fairly easy to adapt the reader and punch to be disk read and write, but those actually took more effort. While the reader and punch functions map to FDOS disk read and write, there were still the file open and close functions that needed to be shoehorned in somewhere.

The Teletype paper tape reader and punch use the same serial lines as the keyboard and printer, so to use them you needed to do something to switch modes. TEA would generate an appropriate prompt to allow the operator to get paper tape loaded in the reader or punch, and make the necessary switch. Those prompts provided a fairly clean way to attach file open functions. There were also equivalent places when TEA was finished reading or punching that could be used for any needed file close functions.

FDOS file open consists of getting a filename, making sure it fits the 6.3 format, checking to see if it exists, and then assigning a buffer. If you are going to read the file it must already exist, obviously, but if you are going to write a file it cannot exist. FDOS does not allow overwriting an existing file. You would need to write to a different file, delete the existing one, or rename it. All of those possibilities had to be covered in the file open functions.

While FDOS requires a file close operation to end a file write, there is no corresponding close for file reads. However, because of the way TEA reads paper tape it was difficult to avoid multiple calls to reader file open. A flag that indicated that a source file was already open was therefore necessary. Clearing that flag is the only thing that the reader's file close function actually does. Besides the original six I/O functions, now there are four more: reader open, reader close, punch open, and punch close.

One thing that I never liked about the TEA assembler was that it used a one byte per line format, instead of the more common one instruction per line. And there were a few other idiosyncrasies that eventually became too annoying, so I wrote an assembler that followed more conventional formatting and syntax. This assembler had the additional advantage of assembling source that was split into multiple files. Once I had TEA working to my satisfaction, I went through the same process with my assembler. This turned out to be easier because I originally wrote it with FDOS in mind, so all I needed to do was verify existing functions, and add some that I had apparently never gotten around to 40 years ago.

At this point I have an OS, auxiliary tools for the OS, a line editor, and two assemblers, all working with the SBC-85 bubble memory board. But there's one more thing to do: make FDOS ROMable. Since the SBC-85 CPU board has a ZIF socket for expansion ROM, I thought it would be a good idea to see if I could get FDOS and its auxiliary programs to run in ROM. At first glance this seems like a fairly easy task: simply put all the code in ROM space, and all the other stuff in RAM. But all of the programs depended in varying degrees on the assembler initializing certain memory locations. So all of those cases had to be tracked down and code had to be added to initialize them properly. And care had to be exercised to make sure that none of the individual programs had any overlaps in RAM.

As a final task, I worked on unifying "hot keys" between all the different programs. They should now all use backspace to delete a previous character, and since you can't actually do a backspace on a Teletype, the previous character is instead echoed with a "\". You can use ^U to cancel an entire line, ^R to retype the line, and "?" for help. TEA does have some single letter commands that could be converted to string form to allow editing, but that's another iten for the future. Although there is certainly plenty more that could be done, all the software parts of this project seem reasonably functional at this point.